Optimal Parameter Selection for Efficient Memory Integrity Verification Using Merkle Hash Trees

Dan Williams and Emin Gün Sirer
{djwill, egs}@cs.cornell.edu
Department of Computer Science
Cornell University
Ithaca, NY 14853

Abstract

A secure, tamperproof execution environment is critical for trustworthy network computing. Newly emerging hardware, such as those developed as part of the TCPA and Palladium initiatives, enables operating systems to implement such an environment through Merkle hash trees. We examine the selection of optimal parameters, namely blocksize and tree depth, for Merkle hash trees based on the size of the memory region to be protected and the number of memory updates between updates of the hash tree. We analytically derive an expression for the cost of updating the hash tree, show that there is an optimal blocksize for the leaves of a Merkle tree for a given filesize and update interval that minimizes the cost of update operations, and describe a general method by which the parameters of such a tree can be determined optimally.

1. Introduction

Trustworthy network computing fundamentally requires the ability to reason about the state of a computation on remote nodes. Such reasoning relies on two mechanisms. First, a node needs to be able to represent the state of a local computation such that other nodes in the network can make an intelligent decision on whether or not to trust the results of that computation. Previous work on attestation [9, 11, 2] addresses precisely this issue; a certificate chain rooted in secure hardware can attest that a given version of the operating system executed a particular version of an application. Using such a certificate chain, a remote game server, for instance, may decide to permit (or reject) a client attempting to connect to the game with a good (or hacked) game client. Similar intelligent trust decisions on whether a client will obey a desired protocol may be made in other distributed computing settings, including peer-to-peer systems and ad hoc networks, using the same mechanism. In essence, the certificate chain can establish that certain predicates over the code, usually represented compactly through code version numbers or cryptographic hashes, hold at a certain point in time.

But attesting to the state of a client at a given point in time is not sufficient to establish trust. A second mechanism, namely, an isolated, secure, tamperproof execution environment, is required to reason about the state of the computation subsequent to the attestation. In the example above, a game server should allow clients to connect only if their binary is verified to not be hacked at the time of connection (achieved through attestation), and if the connected game client can execute in a tamperproof environment where the binary cannot be modified after connection (achieved through tamperproof execution). This latter mechanism has been the subject of much recent work [13, 12, 19, 7, 8], buoyed by the emergence of hardware support for secure execution in general-purpose computers [1] and industry support for secure execution as in Microsoft's Palladium [5]. All mechanisms for tamperproof execution proposed to date rely on costly cryptographic hashes to detect modifications to memory; however, none minimize the cost of hashing.

This paper focuses on the use of cryptographic hashes to secure memory against unauthorized modifications, and derives an expression for optimal hash parameters. A well known method to ensure that the contents of a data structure stored in untrusted storage (memory, disk or tertiary storage) have not been tampered with is to compute a hash of that data upon creation and store the hash in a secure location. The next time an element in the data structure is used, the hash is recomputed and checked against the stored hash; unauthorized modifications to the data structure will be caught through a hash mismatch. However, this naïve use of hashing can become extremely expensive when used on large data structures.

Merkle hash trees have been proposed as a means to reduce the cost for hashing large data structures[15, 16]. They are used to take a secure summary snapshot of a memory region, which can then be used to detect tampering. A memory region is divided into smaller blocks, the hashes of which form the *leaf hashes* at the leaves of a complete binary tree. The value of an inner node of the tree, an *inner hash*, is obtained by concatenating and hashing the values of its child nodes. After a set of updates to a memory region that constitute a transaction, a new secure summary snapshot of the data structure is obtained by incrementally recomputing the leaf hashes corresponding to the modified blocks, as well as the inner hashes from each modified leaf to the root of the tree. Once a new Merkle hash tree is computed, the hashes can be stored in a secure location, such as a secure coprocessor, and used to ascertain the integrity of the data structure kept in ordinary memory. Overall, Merkle hash trees constitute a very simple and effective way to take a secure summary snapshot of a data structure.

The blocksize is the critical parameter of a Merkle hash tree. A large blocksize reduces the depth of the tree at the cost of increasing the leaf hash cost. A small blocksize makes leaf hashes cheaper to compute, though it also increases the depth of the tree, and correspondingly, the time spent computing inner hashes.

This paper examines the optimal selection of blocksize for Merkle hash trees. We derive an analytical model that describes the cost of incremental updates to a Merkle hash tree given the total size of a memory region to be protected and the number of modified memory locations in each transaction, and we can numerically determine the blocksize that minimizes the cost of performing updates to the tree. This, in turn, enables an efficient mechanism for implementing tamperproof execution using commodity memory and storage devices.

This paper makes two contributions. First, it shows that there is a minimum update cost that can be achieved by a hash tree through careful selection of the blocksize at the leaves of the tree. Second, it derives this optimal blocksize given simple parameters, easily determined in practice. The choice of optimal parameters for tamperproof memory in turn leads to efficient systems for secure, trustworthy execution. Surprisingly, the optimal parameters in many common settings differ from natural choices that designers may be tempted to pick, such as the native cacheline and page size.

In the next section, we discuss related work in the areas of tamper-proof memory and Merkle trees. Section 3 describes our system model to help put the problem in context. An analytical model of the problem and results are presented in Section 4. Section 5 discusses the implications of this work for implementing tamperproof execution hardware, the cornerstone of trusted network computing, and Section 6 summarizes the contribution and concludes.

2. Related Work

Merkle trees were originally presented as a method in which two entities can agree on a shared secret using a public key infrastructure [15, 16]. As an alternative to certificate based schemes in which a CA must be trusted to disseminate the correct bindings between an entity and its public key, Merkle proposed that a file containing all the mappings be hashed and the result be widely publicized. If B then wishes to verify that a particular public key corresponds to A, B simply needs to hash all the values of all of the mappings and check that the new hash matches the well known hash. Due to the infeasible requirement of knowledge of every mapping and the need to compute a hash of the potentially enormous file, Merkle suggested that a hash tree be used. A hash tree requires only a few intermediate values on the path from the mapping to verify to the root of the tree in order to reconstruct the hash. These intermediate values are known as the authentication path.

Other applications of Merkle hash trees have been in fast digital signature schemes for flows and multicasts [21] and verification of signatures on read only file systems [6]. There has also been some work focused on the integrity of persistent storage in databases [14] and DRM systems [17].

Blum et al. [3] use Merkle hash trees to provide general memory integrity, in a manner similar to the system model used in this paper. Their analysis shows that $2\log(n)$ cells must be accessed for a read or a write (the cells on the path to the root and the authentication path) and $\log(n)$ hash operations for verification or update of the tree; however, this work does not examine how to determine the hash blocksize.

Recent work on trustworthy execution platforms [13, 12, 18, 7, 19, 5] has examined practical mechanisms for attestation [11, 22, 20] and tamperproof execution. This work spans a large space including the design of secure coprocessors and security enhancements to ordinary processors to provide a trustworthy execution environment, the attestation of the underlying system to the integrity of its applications, the structure of the underlying operating system to provide secure attestation, and finally, on the trustworthiness of applications.

The eXecute Only Memory (XOM) architecture [13] provides a trusted environment for applications through additional hardware in the processor that creates an isolated, secure, tamperproof execution environment to applications. The additional hardware encrypts memory and register values as they are transferred into and out of the processor. This additional hardware enables tamperproof execution guarantees to be provided to applications without having to trust the underlying operating system [12]. XOM, however,

suffers from replay attacks in which data in a compartment can be replaced by old data from that compartment. The memory integrity scheme described in this paper can complement the XOM architecture to efficiently provide tamperproof memory immune to replays.

Terra [7] takes a different approach to trusted execution by providing each application a virtual machine to execute on, managed by a trusted virtual machine monitor. When seeking to verify some amount of data, Terra divides the data into blocks to avoid the high cost of hashing a large object, computes hashes of each block, and then stores the hash of these hashes into the VM descriptor, essentially creating a tree with two levels and a high branching factor. The scheme we propose in this paper can be used to replace the memory integrity scheme used in Terra with an efficient, optimal approach.

AEGIS [19] can run with a security kernel on top of the hardware, similar to Microsoft's Palladium [5], or without trusting the OS, similar to XOM [12]. The memory integrity scheme used by AEGIS is a Merkle hash tree, integrated within the memory hierarchy [8]. AEGIS provides an efficient hardware implementation of Merkle trees by embedding the hash values in processor caches, but does not consider the optimal parameters for the hash tree. Our work can inform architects of secure processors on how to efficiently determine hash block sizes, which interact with the determination of cacheline sizes.

Other techniques have been introduced to provide memory integrity. A fractal-based approach [10] has been proposed to minimize the traversal of a Merkle hash tree; this work also takes the Merkle hash tree as a given and does not examine the selection of blocksize for the hash tree. Incremental multiset hash functions [4] have been proposed as a means to improve memory integrity verification performance by the ability to quickly update read and write logs in trusted storage, to be verified at a later time. This work focuses on sequences of reads and writes, and outperforms a hash tree only in the case of infrequent memory verification.

3. System Model

The motivation for our work comes from the desire to develop a small trustworthy operating system that can provide applications a safe environment in which to execute. We have been building a new operating system, called Nexus, that provides attestation and secure, tamperproof execution based on the TCPA hardware (known as the TPM) [1]. While the design and implementation of this system is beyond the scope of this paper, we outline the system in order to provide a context for the use of Merkle hash trees to provide tamperproof execution.

The Nexus is a secure native operating system that provides trustworthy attestation and tamperproof execution services to its applications. It is arranged as a highly componentized system, where each component operates in a separate, isolated execution environment. The small size of the Nexus reduces the amount of code that operates with system privileges, permits the base system to be audited, and most importantly, enables the principle of least privilege to be used effectively in practice. Whereas in a monolithic operating system, all applications are dependent on, and need to trust, the implementation of all services in the kernel, Nexus applications need to trust only those components that they directly interact with. Figure 1 illustrates the structure of the Nexus.

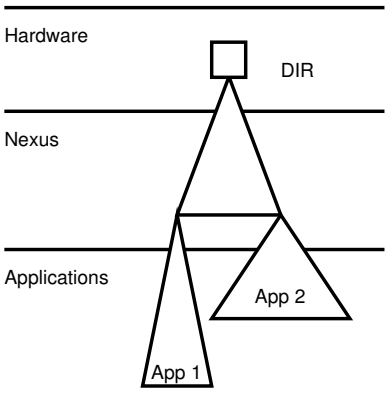

Figure 1. The Nexus provides a protected memory abstraction. Each application may have differently sized protected memory regions with different update characteristics, requiring different configurations of Merkle hash trees for efficient checking. The tree is stored in data integrity registers (DIR) in the trusted coprocessor that is part of the newly emerging TCPA standard.

The Nexus provides interfaces by which secure certificate chains, rooted in the *platform key* embedded in the TPM hardware, can be extended to applications. The platform key is a key embedded by the manufacturer from which other keys can be derived and using which certificate chains can be extended from the boot loader all the way to applications. This, in turn, enables the Nexus to sign certificates that say "The hardware manufacturer attests that it booted this particular version of the Nexus, which attests that it executed this version of the game client." These certificates enable remote nodes to make informed trust decisions.

As discussed before, extending trust based on a certificate requires that the system be capable of retaining predicates established at the time of certificate generation. The

Nexus does this by creating a tamperproof execution environment, where the contents of memory can only be modified by the applications that have been authorized to modify them. The Nexus protects memory regions against tampering by computing a Merkle hash tree over each region and storing parts of the hash tree in the secure TPM hardware. The Nexus provides a very general interface by which applications direct the kernel to create a protected memory region of a given size, using a given blocksize. A toolkit, located in user space, is responsible for determining the optimal blocksize for the Merkle hash tree - thus, the interface is general-purpose, and the complexity of blocksize selection is left out of the kernel. The technique, shown below, is used by the toolkit to determine the optimal blocksize for the Merkle hash tree. We note that many of the other tamperproof execution schemes cited in Section 2 could use the same technique to determine the optimal blocksize in their use of Merkle hash trees.

4. Analytical Model and Results

In this section, we derive an expression for the cost of maintaining a Merkle hash tree in the presence of uniformly distributed updates, and describe a process by which the optimal blocksize (and hence, the depth) of the tree can be determined.

Without loss of generality, consider an application wishing to create and use a tamperproof memory region. We will call the size of this memory the filesize in bytes, denoted by f. We wish to divide the memory region into blocks and build a Merkle hash tree over them in order to achieve an efficient hashing based memory integrity implementation, as shown in Figure 2.

Our goal is to determine the optimal blocksize, b (in bytes), for the leaves of the tree. The Merkle tree constructed on top of the memory region is a complete binary tree, which yields the following expression, where d is the depth of the tree, that relates the blocksize to the size of the memory region and the depth of the tree:

$$f = b\,2^d$$

We assume, without loss of generality, that the memory region can be modified n times between updates to the hash tree that protects the region. n can be conservatively set to one, which will yield a data structure over which the hash tree is recomputed after every modification. In some settings, where the protected data structure in the tamperproof region is being modified as part of a transaction, there may be more than one modification between subsequent recomputations of the hash tree. The use of the n parameter captures such transactions, and $n > 1$ enables performance to be increased where timely updates to the secure summary are not necessary.

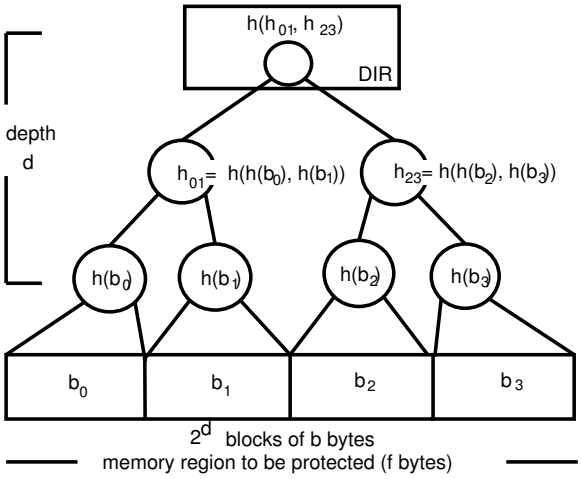

Figure 2. A Merkle tree constructed on top of f bytes of memory using a hash function h.

There are two components contributing to the cost of committing an update to a memory region. First, every leaf responsible for the block on which a modification is made must recompute its hash. Then, every interior node on the path from the affected leaf to the root of the tree must recompute its hash value.

4.1. Cost of Hashing: $H_l(b)$ and H_i

Each of the 2^d leaf nodes in the tree is thus responsible for computing the hash of a block of size b bytes. We model the cost of this hashing operation, referred to as the cost of a leaf hash in μseconds, by:

$$H_l(b) = \alpha b + \beta$$

Our motivation for choosing a linear model for the cost of hashing a data block of size b is based on experimental measurements of the SHA1 hash function performed on an Intel® Pentium® 4 CPU 1700MHz machine, which yielded parameters $\alpha = 0.0122348$ μsec/byte and $\beta = 1$ μsec.

In addition, each of the $2^d - 1$ interior nodes are responsible for hashing the concatenation of the values of its two child nodes. Due to the characteristics of Merkle trees, each of the child nodes contain s bytes, the size of the result of a hash operation. In the case of SHA1, $s = 20$ bytes. Thus the cost (in μsecs) of an inner hash operation is:

$$H_i = 2\alpha s + \beta$$

4.2. Leaf Hash Updates: $U_l(b)$

In order to determine the number of leaf hashes that must be recomputed after n uniformly distributed modifications to a memory region consisting of 2^d blocks, we can first consider the probability of one particular block containing a modification. The probability that the first modification is in a different block is:

$$\frac{2^d - 1}{2^d}$$

The probability that the second modification is also not in our block is equal to the probability that the first modification was not in our particular block multiplied by the probability that the second modification was not in our particular block. Similarly, the probability that all n modifications were not in our one particular block is:

$$\left(\frac{2^d - 1}{2^d}\right)^n$$

On the other hand, the probability that our particular block was touched in one of the n modifications is:

$$1 - \left(\frac{2^d - 1}{2^d}\right)^n$$

Finally, we have 2^d blocks, each with equal probability of being touched, so we can now write the expected number of leaf hashes that need to be recomputed as the expected number of leaf nodes touched after n modifications, or:

$$U_l(b) = 2^d \left(1 - \left(\frac{2^d - 1}{2^d}\right)^n\right)$$

Recall that $f = b\ 2^d$, allowing each d appearing in the right side of the equation to be written in terms of b as $\log_2(\frac{f}{b})$. This substitution has been omitted for clarity.

4.3. Inner Hash Updates: $U_i(b)$

The expected number of inner hashes can be computed in a similar fashion to the leaf hashes. First we consider the inner nodes comprised of the immediate parents of the leaf nodes. Each of these inner nodes has two leaf nodes as its children and will get updated if either of the leaf nodes are updated. So we can think of each inner node on this level responsible for a "block" twice the size of the original blocks, one for each of the memory regions covered by the two child leaf nodes. Then the familiar leaf hash formula will apply and we can see that the number of inner hashes on the lowest level of the tree is:

$$2^{d-1}\left(1 - \left(\frac{2^{d-1} - 1}{2^{d-1}}\right)^n\right)$$

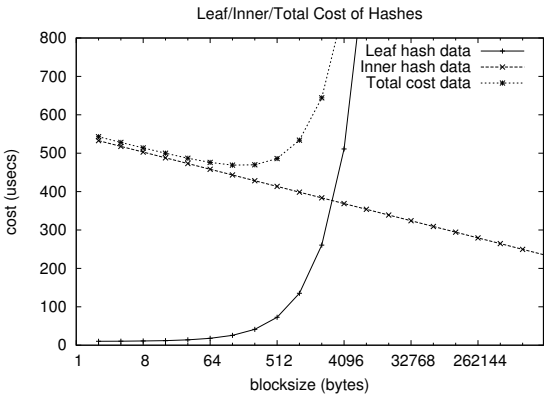

Figure 3. The relationship between the total cost of inner and leaf hashes when the number of updates is $n = 10$, and the filesize is $f = 2^{40}$ bytes. Notice the $\log$ scale on the x axis (blocksize).

A similar argument applies all the way to the root of the tree, which considers the memory region to be two large "blocks". The root node, at depth zero has a probability of 1 that it will be updated as long as $n > 0$. We can write the total number of expected inner hashes that must be performed after n uniformly distributed updates to the memory region as:

$$U_i(b) = \sum_{i=0}^{d-1} 2^i \left(1 - \left(\frac{2^i - 1}{2^i}\right)^n\right)$$

In order to find a closed form for the number of inner hashes, it is useful to notice that the graph of the number of inner hashes, shown in Figure 3, consists of a straight line when viewed in terms of d, a $\log$ in terms of blocksize, followed at some point by a curved tail. Since the linear part of the graph appeared to be the important part of the graph due to the dominance of the cost of leaf hashes as the blocksize increases, we focused on writing an equation for the linear part of the graph.

The following formula depicts the expected number of inner hashes:

$$U_i(b) = nd - \sum_{i=2}^{n} (-1)^i \binom{n}{i} \left(\frac{1}{1 - \frac{1}{2^{i-1}}}\right) + \sum_{i=2}^{n} (-1)^i \binom{n}{i} \left(\frac{1}{1 - \frac{1}{2^{i-1}}}\right) \left(\frac{1}{2^{i-1}}\right)^d$$

In order to make our formula simple, and yet still get an accurate approximation for the number of inner hashes in the tree, we make the observation that the region we are

concerned with is one in which d is relatively large. Thus the terms consisting of fractions raised to a large d become largely unimportant, and we can rewrite the formula for the number of inner hashes as:

$$\overline{U_i}(b) \approx nd - \sum_{i=2}^{n} (-1)^i \binom{n}{i} \left(\frac{1}{1 - \frac{1}{2^{i-1}}} \right)$$

Notice that this is simply nd plus a constant term and be rewritten in terms of b by substituting $\log_2 (\frac{f}{b})$ for d. This accounts for the linear nature of the number of inner hashes as a function of the blocksize on a $\log$ scale.

4.4. Minimizing Cost

The total cost of updates to a Merkle hash tree after n modifications can be computed by combining the formulas for the expected number of leaf hashes and updates to inner nodes. This yields the following:

$$C(b) = \overline{U_i}(b) H_i + U_l(b) H_l(b)$$

Taking the derivative of this function with respect to b can yield the critical points:

$$C'(b) = -\frac{\beta f}{b^2} + \frac{\beta f \left(1 - \frac{b}{f}\right)^n}{b^2} + \alpha n \left(1 - \frac{b}{f}\right)^{n-1} + \frac{\beta n \left(1 - \frac{b}{f}\right)^{n-1}}{b} - \frac{\alpha s n}{b \ln 2} - \frac{\beta n}{b \ln 2}$$

This expression does lend itself readily to an analytical solution. The roots can be determined, however, using a numerical method. We use Newton's method to find the roots of this equation. Newton's method consists of making an estimate x_n for the root, measuring an approximate error term:

$$\epsilon_n = -\frac{C'(x_n)}{C''(x_n)}$$

and updating the current estimate accordingly by adding the error:

$$x_n = x_{n-1} + \epsilon_{n-1}$$

Newton's method has the property of converging quadratically to the minimum, which makes it an acceptable method for finding the minimum of our function.

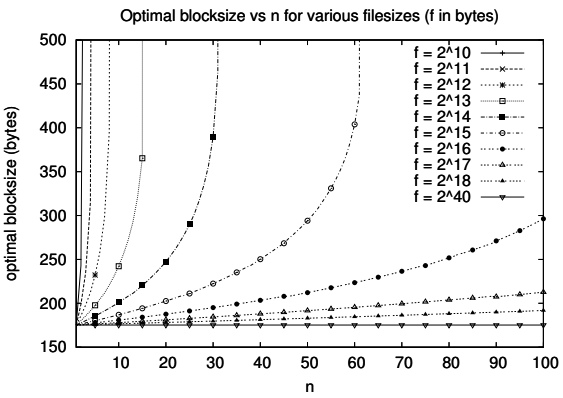

Figure 4. Optimal blocksize in bytes for a variety of file sizes f (bytes) and update intervals n using hash parameters $\alpha = 0.0122348$ μsec/byte, $\beta = 1$ μsec and $s = 20$ bytes.

4.5. Results

The optimal blocksize is dependent on the size of the memory region to be protected (f) and the number of updates (n) to the protected region before the hash tree is updated. Figure 4 illustrates that the optimal blocksize has an asymptote at a constant fraction of the filesize f after which point the optimal blocksize is equal to f, the total filesize. Intuitively, this is the point at which enough blocks have been updated that the overhead of the inner hashes is not worthwhile, and it is faster to collapse the Merkle hash tree down to a single leaf hash over the whole region.

The minimum blocksize is constant between all filesizes because it is dependent only on the parameters of the hash function. To see why this is true, consider one update to a tree with filesize f. The cost of this update, as determined by the cost equation $C(b)$, is:

$$\log_2 \left(\frac{f}{b} \right) H_i + H_l (b)$$

the derivative of which becomes simply:

$$\alpha - \frac{H_i}{b \log (2)}$$

Setting this equation equal to zero, we can solve for:

$$b = \frac{H_i}{\alpha \log (2)}$$

Note that this solution for the optimal blocksize is entirely dependent on the parameters α, β, and s of the hash function. Intuitively, as n increases, the optimal blocksize will increase above this value because each added update could

add d more inner hashes but only one more leaf hash. Increasing the blocksize and thus decreasing the depth of the tree relieves this pressure.

5. Implications

A natural tendency when constructing a Merkle tree is to use architectural constants, such as the native page size of the processor cache line size, for the block size. Figure 4 shows that such quantities often lead to inefficient choices. On our platform, the optimal blocksize is much less than the typical page size for large files, while it is much greater than the typical cache line size for small files. Using the expressions derived above, it is possible to determine the optimum precisely and pick the block size that minimizes the cost of updates to the Merkle hash tree.

5.1. Non-Uniform Distribution of Writes

It is typical for an application to exhibit locality of memory modifications, leading to a distribution of updates that is not uniform. Our analysis has considered update distributions that are uniform. Under a more skewed distribution, we would expect more collisions on the blocks, reducing the leaf hash number, and also earlier collisions as updates propagate up the tree, leading to less inner hashes to compute. This implies that the optimal blocksize curve seen for a particular filesize with respect to n would grow at a slower rate, producing deeper optimal trees.

It is also possible to construct a skewed (non-balanced) Merkle hash tree to reduce the number of inner hashes encountered when updating a very popular region of memory. Ideally, a tree constructed in this way would maintain a low leaf cost due to a small blocksize, and also a low inner hash cost depending on the frequency of memory accesses.

5.2. Caches

Gassend et al. use blocks in the L2 cache as the size of an inner hash so that they can easily cache the inner nodes of the hash tree and then on only worry about propagating the update up to the first inner node contained in the cache[8]. They experiment with two sized L2 block sizes (64B and 128B) in various sized L2 caches and note that increasing the size of the L2 cache block reduces the verification cost by shrinking the tree and increasing its branching factor. However, if the branching factor grows very large, the cost of computing the inner hashes can become a large factor, essentially increasing the y-intercept of the line representing the cost of inner hashes, pushing the optimal blocksize larger.

Thus, even in a situation where the inner hashes are being cached and the branching factor is chosen to improve the performance of the cache, it is still necessary to determine the optimal blocksize given the chosen branching factor of the tree. Our framework is general enough to discover the optimal blocksize for an m-ary tree simply by modifying the relationship between filesize and depth accordingly to $f = b\,m^d$ and the inner hash cost to $H_i = m\alpha s + \beta$.

6. Summary

We have considered the problem of implementing a tamperproof memory region abstraction through the use of one-way hash functions. We examine the use of Merkle hash trees, whose simplicity makes them ideally suited for an efficient implementation. Yet the choice of natural parameters for hash trees, such as the native cacheline size or the native page size, often lead to inefficiencies and excessive costs when recomputing the Merkle hash tree.

This paper analytically derives an expression for the cost of updating the tree, shows that there is an optimal size given a particular combination of filesize and number of memory locations affected by each transaction, and develops a numerical technique for finding the blocksize that optimizes the cost of maintaining the tree. This work is directly applicable to the design of operating system mechanisms, as well as hardware techniques, for providing tamperproof memory. We hope that an analysis of the optimal parameter selection for increasingly ubiquitous Merkle hash trees will enable the newly available trusted hardware to be used to its full potential.

Acknowledgements

We would like to thank Fred B. Schneider for encouraging us to consider a flexible, general-purpose interface for creating protected memory regions.

References

[1] T. C. P. Alliance. Main Specification, Version 1.1a, November 2001.

[2] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[3] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the Correctness of Memories. In *Proceedings of the 32nd annual symposium on Foundations of computer science*, pages 90–99. IEEE Computer Society Press, 1991.

[4] D. Clarke, S. Devadas, B. Gassend, M. van Dijk, and E. Suh. Incremental Multiset Hashes and their Application to Integrity Checking. In *Proceedings of the ASIACRYPT 2003 Conference*, Nov. 2003.

[5] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A Trusted Open Platform. *Computer*, 36(7):55–62, July 2003.

[6] K. Fu, M. F. Kaashoek, and D. Mazieres. Fast and Secure Distributed Read-only File System. *ACM Transactions on Computer Systems*, 20(1):1–24, 2002.

[7] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of the 19th Symposium on Operating System Principles*, Oct. 2003.

[8] B. Gassend, D. Clarke, G. E. Suh, M. van Dijk, and S. Devadas. Caches and Hash Trees for Efficient Memory Integrity Verification. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, Feburary 2003.

[9] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. Distributed System Security Architecture. In *Proceedings of the 12th NIST-NCSC National Computer Security Conference*, pages 305–319, 1989.

[10] M. Jakobsson, T. Leighton, S. Micali, and M. Szydlo. Fractal Merkle Tree Representation and Traversal. In *RSA-CT*, 2003.

[11] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

[12] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 178–192. ACM Press, 2003.

[13] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.

[14] U. Maheshwari, R. Vingralek, and W. Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 135–150, 2000.

[15] R. C. Merkle. Protocols for Public Key Cryptosystems. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[16] R. C. Merkle. A Certified Digital Signature. In *Proceedings on Advances in cryptology*, pages 218–238. Springer-Verlag New York, Inc., 1989.

[17] W. Shapiro and R. Vingralek. How to Manage Persistent State in DRM Systems. In *Digital Rights Management Workshop*, pages 176–191, 2001.

[18] S. W. Smith and S. Weingart. Building a High-Performance, Programmable Secure Coprocessor, April 1999.

[19] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Aegis: Architecture for Tamper-evident and Tamper-resistant Processing, June 2003.

[20] J. Tygar and B. Yee. Dyad: A System for Using Physically Secure Coprocessors. Technical Report Technical Report CMU–CS–91–140R, Carnegie Mellon University, May 1991.

[21] C. K. Wong and S. S. Lam. Digital signatures for flows and multicasts. *IEEE/ACM Trans. Netw.*, 7(4):502–513, 1999.

[22] B. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, May 1994.